



# Money Transfer Made Simple: a Specification, a Generic Algorithm, and its Proof

Alex Auvolat, Davide Frey, Michel Raynal, François Taïani

## ► To cite this version:

Alex Auvolat, Davide Frey, Michel Raynal, François Taïani. Money Transfer Made Simple: a Specification, a Generic Algorithm, and its Proof. Bulletin- European Association for Theoretical Computer Science, 2020, 132. hal-02861511v3

**HAL Id: hal-02861511**

**<https://hal.science/hal-02861511v3>**

Submitted on 16 Feb 2021

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Money Transfer Made Simple: a Specification, a Generic Algorithm, and its Proof

Alex Auvolat<sup>◇,†</sup>, Davide Frey<sup>†</sup>, Michel Raynal<sup>†,\*</sup>, François Taïani<sup>†</sup>

<sup>◇</sup>École Normale Supérieure, Paris, France

<sup>†</sup>Univ Rennes, Inria, CNRS, IRISA, 35000 Rennes, France

<sup>\*</sup>Department of Computing, Polytechnic University, Hong Kong

## Abstract

It has recently been shown that, contrarily to a common belief, money transfer in the presence of faulty (Byzantine) processes does not require strong agreement such as consensus. This article goes one step further: namely, it first proposes a non-sequential specification of the money-transfer object, and then presents a generic algorithm based on a simple FIFO order between each pair of processes that implements it. The genericity dimension lies in the underlying reliable broadcast abstraction which must be suited to the appropriate failure model. Interestingly, whatever the failure model, the money transfer algorithm only requires adding a single sequence number to its messages as control information. Moreover, as a side effect of the proposed algorithm, it follows that money transfer is a weaker problem than the construction of a safe/regular/atomic read/write register in the asynchronous message-passing crash-prone model.

**Keywords:** Asynchronous message-passing system, Byzantine process, Distributed computing, Efficiency, Fault tolerance, FIFO message order, Modularity, Money transfer, Process crash, Reliable broadcast, Simplicity.

## 1 Introduction

**Short historical perspective** Like field-area or interest-rate computations, money transfers have had a long history (see e.g., [21, 27]). Roughly speaking, when looking at money transfer in today's digital era, the issue consists in building a software object that associates an account with each user and provides two operations, one that allows a process to transfer money from one account to another and one that allows a process to read the current value of an account.

The main issue of money transfer lies in the fact that the transfer of an amount of money  $v$  by a user to another user is conditioned to the current value of the former user's account being at least  $v$ . A violation of this condition can lead to the problem of double spending (i.e., the use of the same money more than once), which occurs in the presence of dishonest processes. Another important issue of money transfer resides in the privacy associated with money accounts. This means that a full solution to money transfer must address two orthogonal issues: synchronization (to guarantee the consistency of the money accounts) and confidentiality/security (usually solved with cryptography techniques). Here, like in closely related work [14], we focus on synchronization.

Fully decentralized electronic money transfer was introduced in [25] with the *Bitcoin* cryptocurrency in which there is no central authority that controls the money exchanges issued by users. From a software point of view, Bitcoin adopts a peer-to-peer approach, while from an application point of view it seems to have been motivated by the 2008 subprime crisis [32].

To attain its goal Bitcoin introduced a specific underlying distributed software technology called *blockchain*, which can be seen as a specific distributed state-machine-replication technique, the aim of which is to provide its users with an object known as a concurrent *ledger*. Such an object is defined by two operations, one that appends a new item in such a way that, once added, the item cannot be removed, and a second operation that atomically reads the full list of items currently appended. Hence, a ledger builds a total order on the invocations of its operations. When looking at the synchronization power provided by a ledger in the presence of failures, measured with the consensus-number lens, it has been shown that the synchronization power of a ledger is  $+\infty$  [13, 30]. In a very interesting way, recent work [14] has shown that, in a context where each account has a single owner who can spend the money currently in his/her account, the consensus number of the *money-transfer* concurrent object is 1. An owner is represented by a process in the following.

This is an important result, as it shows that the power of blockchain technology is much stronger (and consequently more costly) than necessary to implement money transfer<sup>1</sup>. To illustrate this discrepancy, considering a sequential specification of the money transfer object, the authors of [14] show first that, in a failure-prone shared-memory system, money transfer can be implemented on top of a snapshot object [1] (whose consensus number is 1, and consequently can be implemented on top of read/write atomic registers). Then, they appropriately modify their shared-memory algorithm to obtain an algorithm that works in asynchronous failure-prone message-passing systems. To allow the processes to correctly validate the money transfers, the resulting algorithm demands them to capture the causality relation linking money transfers and requires each message to carry control information encoding the causal past of the money transfer it carries.

**Content of the article** The present article goes even further. It first presents a non-sequential specification of the money transfer object<sup>2</sup>, and then shows that, contrarily to what is currently accepted, the implementation of a money transfer object does not require the explicit capture of the causality relation linking individual money transfers. To this end, we present a surprisingly simple yet efficient and generic money-transfer algorithm that relies on an underlying reliable-broadcast abstraction. It is efficient as it only requires a very small amount of meta-data in its messages: in addition to money-transfer data, the only control information carried by the messages of our algorithm is reduced to a single sequence number. It is generic in the sense that it can accommodate different failure models *with no modification*. More precisely, our algorithm inherits the fault-tolerance properties of its underlying reliable broadcast: it tolerates crashes if used with a crash-tolerant reliable broadcast, and Byzantine faults if used with a Byzantine-tolerant reliable broadcast.

Given an  $n$ -process system where at most  $t$  processes can be faulty, the proposed algorithm works for  $t < n$  in the crash failure model, and  $t < n/3$  in the Byzantine failure model. This has an interesting side effect on the distributed computability side. Namely, in the crash failure model, money transfer constitutes a weaker problem than the construction of a safe/regular/atomic read/write register (where “weaker” means that—unlike a read/write register—it does not require the “majority of non-faulty processes” assumption).

**Roadmap** The article consists of 7 sections. First, Section 2 introduces the distributed failure-prone computing models in which we are interested, and Section 3 provides a definition of money transfer suited to these computing models. Then, Section 4 presents a very simple generic money-transfer algorithm. Its instantiations and the associated proofs are presented in Section 5 for the crash failure model and in Section 6 for the Byzantine failure model. Finally, Section 7 concludes the article.<sup>3</sup>

---

<sup>1</sup>As far as we know, the fact that consensus is not necessary to implement money transfer was stated for the first time in [15].

<sup>2</sup>To our knowledge, this is the first non-sequential specification of the money transfer object proposed so far.

<sup>3</sup>Let us note that similar ideas have been developed concomitantly and independently in [10], which presents a money transfer system and its experimental evaluation.

## 2 Distributed Computing Models

### 2.1 Process failure model

**Process model** The system comprises a set of  $n$  sequential asynchronous processes, denoted  $p_1, \dots, p_n$ <sup>4</sup>. Sequential means that a process invokes one operation at a time, and asynchronous means that each process proceeds at its own speed, which can vary arbitrarily and always remains unknown to the other processes.

Two process failure models are considered. The model parameter  $t$  denotes an upper bound on the number of processes that can be faulty in the considered model. Given an execution  $r$  (run) a process that commits failures in  $r$  is said to be faulty in  $r$ , otherwise it is non-faulty (or correct) in  $r$ .

**Crash failure model** In this model, processes may crash. A crash is a premature definitive halt. This means that, in the crash failure model, a process behaves correctly (i.e., executes its algorithm) until it possibly crashes. This model is denoted  $CAMP_{n,t}[\emptyset]$  (*Crash Asynchronous Message Passing*). When  $t$  is restricted not to bypass a bound  $f(n)$ , the corresponding restricted failure model is denoted  $CAMP_{n,t}[t \leq f(n)]$ .

**Byzantine failure model** In this model, processes can commit Byzantine failures [23, 28], and those that do so are said to be Byzantine. A Byzantine failure occurs when a process does not follow its algorithm. Hence a Byzantine process can stop prematurely, send erroneous messages, send different messages to distinct processes when it is assumed to send the same message, etc. Let us also observe that, while a Byzantine process can invoke an operation which generates application messages<sup>5</sup> it can also “simulate” this operation by sending fake implementation messages that give their receivers the illusion that they have been generated by a correct sender. However, we assume that there is no Sybil attack like most previous work on byzantine fault tolerance including [14].<sup>6</sup>

As previously, the notations  $BAMP_{n,t}[\emptyset]$  and  $BAMP_{n,t}[t \leq f(n)]$  (*Byzantine Asynchronous Message Passing*) are used to refer to the corresponding Byzantine failure models.

### 2.2 Underlying complete point-to-point network

The set of processes communicate through an underlying message-passing point-to-point network in which there exists a bidirectional channel between any pair of processes. Hence, when a process receives a message, it knows which process sent this message. For simplicity, in writing the algorithms, we assume that a process can send messages to itself.

Each channel is reliable and asynchronous. Reliable means that a channel does not lose, duplicate, or corrupt messages. Asynchronous means that the transit delay of each message is finite but arbitrary. Moreover, in the case of the Byzantine failure model, a Byzantine process can read the content of the messages exchanged through the channels, but cannot modify their content.

To make our algorithm as generic and simple as possible, Section 4 does not present it in terms of low-level send/receive operations<sup>7</sup> but in terms of a high-level communication abstraction, called *reliable broadcast* (e.g., [7, 9, 16, 19, 30]). The definition of this communication abstraction appears in Section 5 for the crash failure model and Section 6 for the Byzantine failure model. It is important to

---

<sup>4</sup>Hence the system we consider is static (according to the distributed computing community parlance) or permissioned (according to the blockchain community parlance).

<sup>5</sup>An *application* message is a message sent at the application level, while an *implementation* is low level message used to ensure the correct delivery of an application message.

<sup>6</sup>As an example, a Byzantine process can neither spawn new identities, nor assume the identity of existing processes.

<sup>7</sup>Actually the send and receive operations can be seen as “machine-level” instructions provided by the network.

note that the previously cited reliable broadcast algorithms do not use sequence numbers. They only use different types of implementation messages which can be encoded with two bits.

### 3 Money Transfer: a Formal Definition

**Money transfer: operations** From an abstract point of view, a money-transfer object can be seen as an abstract array  $ACCOUNT[1..n]$  where  $ACCOUNT[i]$  represents the current value of  $p_i$ 's account. This object provides the processes with two operations denoted  $balance()$  and  $transfer()$ , whose semantics are defined below. The transfer by a process of the amount of money  $v$  to a process  $p_j$  is represented by the pair  $\langle j, v \rangle$ . Without loss of generality, we assume that a process does not transfer money to itself. It is assumed that each  $ACCOUNT[i]$  is initialized to a non-negative value denoted  $init[i]$ . It is assumed the array  $init[1..n]$  is initially known by all the processes.<sup>8</sup>

Informally, when  $p_i$  invokes  $balance(j)$  it obtains a value (as defined below) of  $ACCOUNT[j]$ , and when it invokes the transfer  $\langle j, v \rangle$ , the amount of money  $v$  is moved from  $ACCOUNT[i]$  to  $ACCOUNT[j]$ . If the transfer succeeds, the operation returns `commit`, if it fails it returns `abort`.

**Histories** The following notations and definitions are inspired from [2].

- A local execution history (or local history) of a process  $p_i$ , denoted  $L_i$ , is a sequence of operations  $balance()$  and  $transfer()$  issued by  $p_i$ . If an operation  $op1$  precedes an operation  $op2$  in  $L_i$ , we say that “ $op1$  precedes  $op2$  in process order”, which is denoted  $op1 \rightarrow_i op2$ .
- An execution history (or history)  $H$  is a set of  $n$  local histories, one per process,  $H = (L_1, \dots, L_n)$ .
- A serialization  $S$  of a history  $H$  is a sequence that contains all the operations of  $H$  and respects the process order  $\rightarrow_i$  of each process  $p_i$ .
- Given a history  $H$  and a process  $p_i$ , let  $A_{i,T}(H)$  denote the history  $(L'_1, \dots, L'_n)$  such that
  - $L'_i = L_i$ , and
  - For any  $j \neq i$ :  $L'_j$  contains only the transfer operations of  $p_j$ .

#### Notations

- An operation  $transfer(j, v)$  invoked by  $p_i$  is denoted  $trf_i(j, v)$ .
- An invocation of  $balance(j)$  that returns the value  $v$  is denoted  $blc(j)/v$ .
- Let  $H$  be a set of operations.
  - $plus(j, H) = \sum_{trf_k(j,v) \in H} v$  (total of the money given to  $p_j$  in  $H$ ).
  - $minus(j, H) = \sum_{trf_j(k,v) \in H} v$  (total of the money given by  $p_j$  in  $H$ ).
  - $acc(j, H) = init[j] + plus(j, H) - minus(j, H)$  (value of  $ACCOUNT[j]$  according to  $H$ ).
- Given a history  $H$  and a process  $p_i$ , let  $S_i$  be a serialization of  $A_{i,T}(H)$  (hence,  $S_i$  respects the  $n$  process orders defined by  $H$ ). Let  $\rightarrow_{S_i}$  denote the total order defined by  $S_i$ .

**Money-transfer-compliant serialization** A serialization  $S_i$  of  $A_{i,T}(H)$  is money-transfer compliant (MT-compliant) if:

- For any operation  $trf_j(k, v) \in S_i$ , we have
 
$$v \leq acc(j, \{op \in S_i \mid op \rightarrow_{S_i} trf_j(k, v)\}),$$
- For any operation  $blc(j)/v \in S_i$ , we have
 
$$v = acc(j, \{op \in S_i \mid op \rightarrow_{S_i} blc(j)/v\}).$$

---

<sup>8</sup>It is possible to initialize some accounts to negative values. In this case, we must assume  $pos > neg$ , where  $pos$  (resp.,  $neg$ ) is the sum of all the positive (resp., negative) initial values.

MT-compliance is the key concept at the basis of the definition of a money-transfer object. It states that it is possible to associate each process  $p_i$  with a total order  $S_i$  in which (a) each of its invocations of  $\text{balance}(j)$  returns a value  $v$  equal to  $p_j$ 's account's current value according to  $S_i$ , and (b) processes transfer only money that they have.

Let us observe that the common point among the serializations  $S_1, \dots, S_n$  lies in the fact that each process sees all the transfer operations of any other process  $p_j$  in the order they have been produced (as defined by  $L_j$ ), and sees its own transfer and balance operations in the order it produced them (as defined by  $L_i$ ).

**Money transfer in  $CAMP_{n,t}[\emptyset]$**  Considering the  $CAMP_{n,t}[\emptyset]$  model, a money-transfer object is an object that provides the processes with  $\text{balance}()$  and  $\text{transfer}()$  operations and is such that, for each of its executions, represented by the corresponding history  $H$ , we have:

- All the operations invoked by correct processes terminate.
- For any correct process  $p_i$ , there is an MT-compliant serialization  $S_i$  of  $A_{i,T}(H)$ , and
- For any faulty process  $p_i$ , there is a history  $H' = (L'_1, \dots, L'_n)$  where (a)  $L'_j$  is a prefix of  $L_j$  for any  $j \neq i$ , and (b)  $L'_i = L_i$ , and there is an MT-compliant serialization of  $A_{i,T}(H')$ .

An algorithm implementing a money transfer object is correct in  $CAMP_{n,t}[\emptyset]$  if it produces only executions as defined above. We then say that the algorithm is MT-compliant.

**Money transfer in  $BAMP_{n,t}[\emptyset]$**  The main differences between money transfer in  $CAMP_{n,t}[\emptyset]$  and  $BAMP_{n,t}[\emptyset]$  lies in the fact that a faulty process can try to transfer money it does not have, and try to present different behaviors with respect to different correct processes. This means that, while the notion of a local history  $L_i$  is still meaningful for a non-Byzantine process, it is not for a Byzantine process. For a Byzantine process, we therefore define a *mock local history* for a process  $p_i$  as any sequence of transfer operations from  $p_i$ 's account<sup>9</sup>. In this definition, the mock local history  $L_i$  associated with a Byzantine process  $p_i$  is not necessarily the local history it produced, it is only a history that it could have produced from the point of view of the correct processes. The correct processes implement a money-transfer object if they all behave in a manner consistent with the same set of mock local histories for the Byzantine processes. More precisely, we define a *mock history* associated with an execution on a money transfer object in  $BAMP_{n,t}[\emptyset]$  as  $\tilde{H} = (\tilde{L}_1, \dots, \tilde{L}_n)$  where:

$$\tilde{L}_j = \begin{cases} L_j & \text{if } p_j \text{ is correct,} \\ \text{a mock local history} & \text{if } p_j \text{ is Byzantine.} \end{cases}$$

Considering the  $BAMP_{n,t}[\emptyset]$  model, a money transfer object is such that, for each of its executions, there exists a *mock history*  $\tilde{H}$  such that for any correct process  $p_i$ , there is an MT-compliant serialization  $S_i$  of  $A_{i,T}(\tilde{H})$ . An algorithm implementing such executions is said to be MT-compliant.

**Concurrent vs sequential specification** Let us notice that the previous specification considers money transfer as a concurrent object. More precisely and differently from previous specifications of the money transfer object, it does not consider it as a sequential object for which processes must agree on the very same total order on the operations they issue [17]. As a simple example, let us consider two processes  $p_i$  and  $p_j$  that independently issue the transfers  $\text{trf}_i(k, v)$  and  $\text{trf}_j(k, v')$  respectively. The proposed specification allows these transfers (and many others) to be seen in different order by different processes. As far as we know, this is the first specification of money transfer as a non-sequential object.

---

<sup>9</sup>Let us remind that the operations  $\text{balance}()$  issued by a Byzantine can return any value. So they are not considered in the mock histories associated with Byzantine processes.

## 4 A Simple Generic Money Transfer Algorithm

This section presents a generic algorithm implementing a money transfer object. As already said, its generic dimension lies in the underlying reliable broadcast abstraction used to disseminate money transfers to the processes, which depends on the failure model.

### 4.1 Reliable broadcast

Reliable broadcast provides two operations denoted `r_broadcast()` and `r_deliver()`. Because a process is assumed to invoke reliable broadcast each time it issues a money transfer, we use a *multi-shot* reliable broadcast, that relies on *explicit sequence numbers* to distinguish between its different instances (more on this below). Following the parlance of [16] we use the following terminology: when a process invokes `r_broadcast( $sn, m$ )`, we say it “r-broadcasts the message  $m$  with sequence number  $sn$ ”, and when its invocation of `r_deliver()` returns it a pair  $(sn, m)$ , we say it “r-delivers  $m$  with sequence number  $sn$ ”. While definitions of reliable broadcast suited to the crash failure model and the Byzantine failure model will be given in Section 5 and Section 6, respectively, we state their common properties below.

- **Validity.** This property states that there is no message creation. To this end, it relates the outputs (r-deliveries) to the inputs (r-broadcasts). Excluding malicious behaviors, a message that is r-delivered has been r-broadcast.
- **Integrity.** This property states that there is no message duplication.
- **Termination-1.** This property states that correct processes r-deliver what they broadcast.
- **Termination-2.** This property relates the sets of messages r-delivered by different processes.

The Termination properties ensure that all the correct processes r-deliver the same set of messages, and that this set includes at least all the messages that they r-broadcast.

As mentioned above, sequence numbers are used to identify different instances of the reliable broadcast. Instead of using an underlying FIFO-reliable broadcast in which sequence numbers would be hidden, we expose them in the input/output parameters of the `r_broadcast()` and `r_deliver()` operations, and handle their updates explicitly in our generic algorithm. This reification<sup>10</sup> allows us to capture explicitly the complete control related to message r-deliveries required by our algorithm. As we will see, it follows that the instantiations of the previous Integrity property (crash and Byzantine models) will explicitly refer to “upper layer” sequence numbers.

We insist on the fact that the reliable broadcast abstraction that the proposed algorithm depends on does not itself provide the FIFO ordering guarantee. It only uses sequence numbers to identify the different messages sent by a process. As explained in the next section, the proposed generic algorithm implements itself the required FIFO ordering property.

### 4.2 Generic money transfer algorithm: local data structures

As said in the previous section, `init[1..n]` is an array of constants, known by all the processes, such that `init[k]` is the initial value of  $p_k$ ’s account, and a transfer of the quantity  $v$  from a process  $p_i$  to a process  $p_k$  is represented by the pair  $\langle k, v \rangle$ . Each process  $p_i$  manages the following local variables:

- $sn_i$ : integer variable, initialized to 0, used to generate the sequence numbers associated with the transfers issued by  $p_i$  (it is important to notice that the point-to-point FIFO order realized with the sequence numbers is the only “causality-related” control information used in the algorithm).
- $del_i[1..n]$ : array initialized to  $[0, \dots, 0]$  such that  $del_i[j]$  is the sequence number of the last transfer issued by  $p_j$  and locally processed by  $p_i$ .

---

<sup>10</sup>Reification is the process by which an implicit, hidden or internal information is explicitly exposed to a programmer.

- $account_i[1..n]$ : array, initialized to  $init[1..n]$ , that is a local approximate representation of the abstract array  $ACCOUNT[1..n]$ , i.e.,  $account_i[j]$  is the value of  $p_j$ 's account, as known by  $p_i$ .

While other local variables containing bookkeeping information can be added according to the application's needs, it is important to insist on the fact that the proposed algorithm needs only the three previous local variables (i.e.,  $(2n + 1)$  local registers) to solve the synchronization issues that arise in fault-tolerant money transfer.

### 4.3 Generic money transfer algorithm: behavior of a process $p_i$

Algorithm 1 describes the behavior of a process  $p_i$ . When it invokes  $balance_i(j)$ ,  $p_i$  returns the current value of  $account_i[j]$  (line 1).

```

init:  $account_i[1..n] \leftarrow init[1..n]$ ;  $sn_i \leftarrow 0$ ;  $del_i[1..n] \leftarrow [0, \dots, 0]$ .

operation  $balance(j)$  is
(1)   $return(account[j])$ .

operation  $transfer(j, v)$  is
(2)  if  $(v \leq account_i[i])$ 
(3)    then  $sn_i \leftarrow sn_i + 1$ ;  $r\_broadcast(sn_i, TRANSFER\langle j, v \rangle)$ ;
(4)     $wait(del_i[i] = sn_i)$ ;  $return(commit)$ 
(5)    else  $return(abort)$ 
(6)  end if.

when  $(sn, TRANSFER\langle k, v \rangle)$  is  $r\_delivered$  from  $p_j$  do
(7)   $wait((sn = del_i[j] + 1) \wedge (account_i[j] \geq v))$ ;
(8)   $account_i[j] \leftarrow account_i[j] - v$ ;  $account_i[k] \leftarrow account_i[k] + v$ ;
(9)   $del_i[j] \leftarrow sn$ .

```

Algorithm 1: Generic broadcast-based money transfer algorithm (code for  $p_i$ )

When it invokes  $transfer(j, v)$ ,  $p_i$  first checks if it has enough money in its account (line 2) and returns `abort` if it does not (line 5). If process  $p_i$  has enough money, it computes the next sequence number  $sn_i$  and  $r$ -broadcasts the pair  $(sn_i, TRANSFER\langle j, v \rangle)$  (line 3). Then  $p_i$  waits until it has locally processed this transfer (lines 7-9), and finally returns `commit`. Let us notice that the predicate at line 7 is always satisfied when  $p_i$   $r$ -delivers a transfer message it has  $r$ -broadcast.

When  $p_i$   $r$ -delivers a pair  $(sn, TRANSFER\langle k, v \rangle)$  from a process  $p_j$ , it does not process it immediately. Instead,  $p_i$  waits until (i) this is the next message it has to process from  $p_j$  (to implement FIFO ordering) and (ii) its local view of the money transfers to and from  $p_j$  (namely the current value of  $account_i[j]$ ) allows this money transfer to occur (line 7). When this happens,  $p_i$  locally registers the transfer by moving the quantity  $v$  from  $account_i[j]$  to  $account_i[k]$  (line 8) and increases  $del_i[j]$  (line 9).

## 5 Crash Failure Model: Instantiation and Proof

This section presents first the crash-tolerant reliable broadcast abstraction whose operations instantiate the  $r\_broadcast()$  and  $r\_deliver()$  operations used in the generic algorithm. Then, using the MT-compliance notion, it proves that Algorithm 1 combined with a crash-tolerant reliable broadcast implements a money transfer object in  $CAMP_{n,t}[\emptyset]$ . It also shows that, in this model, money transfer is weaker than the construction of an atomic read/write register. Finally, it presents a simple weakening of the FIFO requirement that works in the  $CAMP_{n,t}[\emptyset]$  model.



## 5.1 Multi-shot reliable broadcast abstraction in $CAMP_{n,t}[\emptyset]$

This communication abstraction, named CR-Broadcast, is defined by the two operations `cr_broadcast()` and `cr_deliver()`. Hence, we use the terminology “to cr-broadcast a message”, and “to cr-deliver a message”.

- **CRB-Validity.** If a process  $p_i$  cr-delivers a message with sequence number  $sn$  from a process  $p_j$ , then  $p_j$  cr-broadcast it with sequence number  $sn$ .
- **CRB-Integrity.** For each sequence number  $sn$  and sender  $p_j$  a process  $p_i$  cr-delivers at most one message with sequence number  $sn$  from  $p_j$ .
- **CRB-Termination-1.** If a correct process cr-broadcasts a message, it cr-delivers it.
- **CRB-Termination-2.** If a process cr-delivers a message from a (correct or faulty) process  $p_j$ , then all correct processes cr-deliver it.

CRB-Termination-1 and CRB-Termination-2 capture the “strong” reliability property of CR-Broadcast, namely: all the correct processes cr-deliver the same set  $S$  of messages, and this set includes at least the messages they cr-broadcast. Moreover, a faulty process cr-delivers a subset of  $S$ . Algorithms implementing the CR-Broadcast abstraction in  $CAMP_{n,t}[\emptyset]$  are described in [16, 30].

## 5.2 Proof of the algorithm in $CAMP_{n,t}[\emptyset]$

**Lemma 1.** *Any invocation of `balance()` or `transfer()` issued by a correct process terminates.*

**Proof** The fact that any invocation of `balance()` terminates follows immediately from the code of the operation.

When a process  $p_i$  invokes `transfer( $j, v$ )`, it r-broadcasts a message and, due to the CRB-Termination properties,  $p_i$  receives its own transfer message and the predicate (line 7) is necessarily satisfied. This is because (i) only  $p_i$  can transfer its own money, (ii) the wait statement of line 4 ensures the current invocation of `transfer( $j, v$ )` does not return until the corresponding TRANSFER message is processed at lines 8-9, and (iii) the fact that  $account_i[i]$  cannot decrease between the execution of line 3 and the one of line 7. It follows that  $p_i$  terminates its invocation of `transfer( $j, v$ )`.  $\square$  Lemma 1

The safety proof is more involved. It consists in showing that any execution satisfies MT-compliance as defined in Section 3.

### Notation and definition

- Let  $trf_j^{sn}(k, v)$  denote the operation `trf( $k, v$ )` issued by  $p_j$  with sequence number  $sn$ .
- We say a process  $p_i$  *processes* the transfer  $trf_j^{sn}(k, v)$  if, after it cr-delivered the associated message `TRANSFER( $k, v$ )` with sequence number  $sn$ ,  $p_j$  exits the wait statement at line 7 and executes the associated statements at lines 8-9. The moment at which these lines are executed is referred to as the *moment when the transfer is processed* by  $p_i$ . (These notions are related to the progress of processes.)
- If the message TRANSFER cr-broadcast by a process is cr-delivered by a correct process, we say that the transfer is *successful*. (Let us notice that a message cr-broadcast by a correct process is always successful.)

**Lemma 2.** *If a process  $p_i$  processes  $trf_\ell^{sn}(k, v)$ , then any correct process processes it.*

**Proof** Let  $m_1, m_2, \dots$  be the sequence of transfers processed by  $p_i$  and let  $p_j$  be a correct process. We show by induction on  $z$  that, for all  $z$ ,  $p_j$  processes all the messages  $m_1, m_2, \dots, m_z$ .

Base case  $z = 0$ . As the sequence of transfers is empty, the proposition is trivially satisfied.

Induction. Taking  $z \geq 0$ , suppose  $p_j$  processed all the transfers  $m_1, m_2, \dots, m_z$ . We have to show that  $p_j$  processes  $m_{z+1}$ . Note that  $m_1, m_2, \dots, m_z$  do not typically originate from the same sender, and are therefore normally processed by  $p_j$  in a different order than  $p_i$ , possibly mixed with other messages. This also applies to  $m_{z+1}$ . If  $m_{z+1}$  was processed by  $p_j$  before  $m_z$ , we are done. Otherwise there is a time  $\tau$  at which  $p_j$  processed all the transfers  $m_1, m_2, \dots, m_z$  (case assumption), cr-delivered  $m_{z+1}$  (CBR-Termination-2 property), but has not yet processed  $m_{z+1}$ . Let  $m_{z+1} = \text{trf}_\ell^{sn}(k, v)$ . At time  $\tau$ , we have the following.

- On one side,  $\text{del}_j[\ell] \leq sn - 1$  since messages are processed in FIFO order and  $m_{z+1}$  has not yet been processed. On the other side,  $\text{del}_j[\ell] \geq sn - 1$  because either  $sn = 1$  or  $\text{trf}_\ell^{sn-1}(-, -) \in m_1, \dots, m_z$ , where  $\text{trf}_\ell^{sn-1}(-, -)$  is the transfer issued by  $p_\ell$  just before  $m_{z+1} = \text{trf}_\ell^{sn}(k, v)$  (otherwise  $p_i$  would not have processed  $m_{z+1}$  just after  $m_1, \dots, m_z$ ). Thus  $\text{del}_j[\ell] = sn - 1$ .
- Let us now shown that, at time  $\tau$ ,  $\text{account}_j[\ell] \geq v$ . To this end let  $\text{plus}_i^{z+1}(\ell)$  denote the money transferred to  $p_\ell$  as seen by  $p_i$  just before  $p_i$  processes  $m_{z+1}$ , and  $\text{minus}_i^{z+1}(\ell)$  denote the money transferred from  $p_\ell$  as seen by  $p_i$  just before  $p_i$  processes  $m_{z+1}$ . Similarly, let  $\text{plus}_j^{z+1}(\ell)$  denote the money transferred to  $p_\ell$  as seen by  $p_j$  at time  $\tau$  and  $\text{minus}_j^{z+1}(\ell)$  denote the money transferred from  $p_\ell$  as seen by  $p_j$  at time  $\tau$ . Let us consider the following sums:
  - On the side of the money transferred to  $p_\ell$  as seen by  $p_j$ . Due to induction, all the transfers to  $p_\ell$  included in  $m_1, m_2, \dots, m_z$  (and possibly more transfers to  $p_\ell$ ) have been processed by  $p_j$ , thus  $\text{plus}_j^{z+1}(\ell) \geq \sum_{\text{trf}_{k'}(\ell, w) \in \{m_1, m_2, \dots, m_z\}} w$  and, as  $p_i$  processed the messages in the order  $m_1, \dots, m_z, m_{z+1}$  (assumption), we have  $\text{plus}_i^{z+1}(\ell) = \sum_{\text{trf}_{k'}(\ell, w) \in \{m_1, m_2, \dots, m_z\}} w$ . Hence,  $\text{plus}_j^{z+1}(\ell) \geq \text{plus}_i^{z+1}(\ell)$ .
  - On the side of the money transferred from  $p_\ell$  as seen by  $p_j$ . Let us observe that  $p_j$  has processed all the transfers from  $p_\ell$  with a sequence number smaller than  $sn$  and no transfer from  $p_\ell$  with a sequence number greater than or equal to  $sn$ , thus we have  $\text{minus}_j^{z+1}(\ell) = \sum_{\text{trf}_{k'}(k', w) \in \{m_1, m_2, \dots, m_z\}} w = \text{minus}_i^{z+1}(\ell)$ .

Let  $\text{account}_i^{z+1}[\ell]$  be the value of  $\text{account}_i[\ell]$  just before  $p_i$  processes  $m_{z+1}$ , and  $\text{account}_j^{z+1}[\ell]$  be the value of  $\text{account}_j[\ell]$  at time  $\tau$ . As  $\text{account}_j^{z+1}[\ell] = \text{init}[\ell] + \text{plus}_j^{z+1}(\ell) - \text{minus}_j^{z+1}(\ell)$  and  $\text{account}_i^{z+1}[\ell] = \text{init}[\ell] + \text{plus}_i^{z+1}(\ell) - \text{minus}_i^{z+1}(\ell)$ , it follows that  $\text{account}_j[\ell]$  is greater than or equal to the value of  $\text{account}_i[\ell]$  just before  $p_i$  processes  $m_{z+1}$ , which was itself greater than or equal to  $v$  (otherwise  $p_i$  would not have processed  $m_{z+1}$  at that time). It follows that  $\text{account}_j[\ell] \geq v$ .

The two predicates of line 7 are therefore satisfied, and will remain so until  $m_{z+1}$  is processed (due to the FIFO order on transfers issued by  $p_\ell$ ), thus ensuring that process  $p_j$  processes the transfer  $m_{z+1}$ .

□<sub>Lemma 2</sub>

**Lemma 3.** *If a process  $p_i$  issues a successful money transfer  $\text{trf}_i^{sn}(k, v)$  (i.e., it cr-broadcasts it in line 3), any correct process eventually cr-delivers and processes it.*

**Proof** When process  $p_i$  cr-broadcast money transfer  $\text{trf}_i^{sn}(k, v)$ , the local predicate  $(sn = \text{del}_i[i] + 1) \wedge (\text{account}_i[i] \geq v)$  was true at  $p_i$ . When  $p_i$  cr-delivers its own transfer message, the predicate is still true at line 7 and  $p_i$  processes its transfer (if  $p_i$  crashes after having cr-broadcast the transfer and before processing it, we extend its execution—without loss of correctness—by assuming it crashed just after processing the transfer). It follows from Lemma 2 that any correct process processes  $\text{trf}_i^{sn}(k, v)$ .

□<sub>Lemma 3</sub>

**Theorem 1.** *Algorithm 1 instantiated with CR-Broadcast implements a money transfer object in the  $\text{CAMP}_{n,t}[\emptyset]$  system model, and ensures that all operations by correct processes terminate.*

**Proof** Lemma 1 proved that the invocations of the operations `balance()` and `transfer()` by the correct processes terminate. Let us now consider MT-compliance.

Considering any execution of the algorithm, captured as history  $H = (L_1, \dots, L_n)$ , let us first consider a correct process  $p_i$ . Let  $S_i$  be the sequence of the following events happening at  $p_i$  (these events are “instantaneous” in the sense  $p_i$  is not interrupted when it produces each of them):

- the event  $\text{trf}_j^{sn}(k, v)$  (occurs when  $p_j$  broadcasts the corresponding transfer (lines 8-9 executed without interruption).

We show that  $S_i$  is an MT-compliant serialization of  $A_{i,T}(H)$ . When considering the construction of  $S_i$ , we have the following:

- For all  $\text{trf}_j^{sn}(k, v) \in L_j$  we have that  $p_j$  cr-broadcast this transfer and that  $(sn, \text{TRANSFER}\langle k, v \rangle)$  was received by  $p_j$  and was therefore *successful*: it follows from Lemma 3 that  $p_i$  processes this money transfer, and consequently we have  $\text{trf}_j^{sn}(k, v) \in S_i$ .
- For all  $\text{op1} = \text{trf}_j^{sn}(k, v)$  and  $\text{op2} = \text{trf}_j^{sn'}(k', v')$  in  $S_i$  (two transfers issued by  $p_j$ ) such that  $\text{op1} \rightarrow_j \text{op2}$ , we have  $sn < sn'$ . Consequently  $p_i$  processes  $\text{op1}$  before  $\text{op2}$ , and we have  $\text{op1} \rightarrow_{S_i} \text{op2}$ .
- For all pairs  $\text{op1}$  and  $\text{op2}$  belonging to  $L_i$ , their serialization order is the same in  $L_i$  and  $S_i$ .

It follows that  $S_i$  is a serialization of  $A_{i,T}(H)$ . Let us now show that  $S_i$  is MT-compliant.

- Case where the event in  $S_i$  is  $\text{trf}_j^{sn}(k, v)$ . In this case we have  $v \leq \text{acc}(j, \{\text{op} \in S_i \mid \text{op} \rightarrow_{S_i} \text{trf}_j^{sn}(k, v)\})$  because this condition is directly encoded at  $p_i$  in the waiting predicate that precedes the processing of  $\text{op}$ .
- Case where the event in  $S_i$  is  $\text{blc}(j)/v$ . In this case we have  $v = \text{acc}(j, \{\text{op} \in S_i \mid \text{op} \rightarrow_{S_i} \text{blc}(j)/v\})$ , because this is exactly the way how the returned value  $v$  is computed in the algorithm.

This terminates the proof for the correct processes.

For a process  $p_i$  that crashes, the sequence of money transfers from a process  $p_j$  that is processed by  $p_i$  is a prefix of the sequence of money transfers issued by  $p_j$  (this follows from the FIFO processing order, line 7). Hence, for each process  $p_i$  that crashes there is a history  $H' = (L'_1, \dots, L'_n)$  where  $L'_j$  is a prefix of  $L_j$  for each  $j \neq i$  and  $L'_i = L_i$ , such that, following the same reasoning, the construction  $S_i$  given above is an MT-compliant serialization of  $A_{i,T}(H')$ , which concludes the proof of the theorem.

□*Theorem 1*

### 5.3 Money transfer vs read/write registers in $CAMP_{n,t}[\emptyset]$

It is shown in [5] that it is impossible to implement an atomic read/write register in the distributed system model  $CAMP_{n,t}[\emptyset]$ , i.e., when, in addition to asynchrony, any number of processes may crash. On the positive side, several algorithms implementing such a register in  $CAMP_{n,t}[t < n/2]$  have been proposed, each with its own features (see for example [4, 5, 24] to cite a few). An atomic read/write register can be built from safe or regular registers<sup>11</sup> [22, 29, 33]. Hence, as atomic registers, safe and regular registers cannot be built in  $CAMP_{n,t}[\emptyset]$  (although they can in  $CAMP_{n,t}[t < n/2]$ ). As  $CAMP_{n,t}[t < n/2]$  is a more constrained model than  $CAMP_{n,t}[\emptyset]$ , it follows that, from a  $CAMP_{n,t}$  computability point of view, the construction of a safe/regular/atomic read/write register is a stronger problem than money transfer.

<sup>11</sup>Safe and regular registers were introduced introduced in [22]. They have weaker specifications than atomic registers.

## 5.4 Replacing FIFO by a weaker ordering in $CAMP_{n,t}[\emptyset]$

An interesting question is the following one: is FIFO ordering necessary to implement money transfer in the  $CAMP_{n,t}[\emptyset]$  model? While we conjecture it is, it appears that, a small change in the specification of money transfer allows us to use a weakened FIFO order, as shown below.

**Weakened money transfer specification** The change in the specification presented in Section 3 concerns the definition of the serialisation  $S_i$  associated with each process  $p_i$ . In this modified version the serialization  $S_i$  associated with each process  $p_i$  is no longer required to respect the process order on the operations issued by  $p_j$ ,  $j \neq i$ . This means that two different process  $p_i$  and  $p_k$  may observe the  $\text{transfer}()$  operations issued by a process  $p_j$  in different orders (which captures the fact that some transfer operations by a process  $p_j$  are commutative with respect to its current account).

**Modification of the algorithm** Let  $k$  be a constant integer  $\geq 1$ . Let  $sn_i(j)$  be the highest sequence number such that all the transfer messages from  $p_j$  whose sequence numbers belong to  $\{1, \dots, sn_i(j)\}$  have been cr-delivered and processed by a certain process  $p_i$  (i.e., lines 8-9 have been executed for these messages). Initially we have  $sn_i(j) = 0$ .

Let  $sn$  be the sequence number of a message cr-delivered by  $p_i$  from  $p_j$ . At line 7 the predicate  $sn = del_i[j] + 1$  can be replaced by the predicate  $sn \in \{sn_i(j) + 1, \dots, sn_i(j) + k\}$ . Let us notice that this predicate boils down to  $sn = del_i[j] + 1$  when  $k = 1$ . More generally the set of sequence numbers  $\{sn_i(j) + 1, \dots, sn_i(j) + k\}$  defines a sliding window for sequence numbers which allows the corresponding messages to be processed.

The important point here is the fact that messages can be processed in an order that does not respect their sending order as long as all the messages are processed, which is not guaranteed when  $k = +\infty$ . Assuming  $p_j$  issues an infinite number of transfers, if  $k = +\infty$  it is possible that, while all these messages are cr-delivered by  $p_i$ , some of them are never processed at lines 8-9 (their processing being always delayed by other messages that arrive after them). The finiteness of the value  $k$  prevents this unfair message-processing order from occurring.

The proof of Section 5.2 must be appropriately adapted to show that this modification implements the weakened money-transfer specification.

## 6 Byzantine Failure Model: Instantiation and Proof

This section presents first the reliable broadcast abstraction whose operations instantiate the  $r\_broadcast()$  and  $r\_deliver()$  operations used in the generic algorithm. Then, it proves that the resulting algorithm correctly implements a money transfer object in  $BAMP_{n,t}[t < n/3]$ .

### 6.1 Reliable broadcast abstraction in $BAMP_{n,t}[t < n/3]$

The communication abstraction, denoted BR-Broadcast, was introduced in [7]. It is defined by two operations denoted  $br\_broadcast()$  and  $br\_deliver()$  (hence we use the terminology “br-broadcast a message” and “br-deliver a message”). The difference between this communication abstraction and CR-Broadcast lies in the nature of failures. Namely, as a Byzantine process can behave arbitrarily, CRB-Validity, CRB-Integrity, and CRB-Termination-2 cannot be ensured. As an example, it is not possible to ensure that if a Byzantine process br-delivers a message, all correct processes br-deliver it. BR-Broadcast is consequently defined by the following properties. Termination-1 is the same in both communication abstractions, while Integrity, Validity and Termination-2 consider only correct processes (the difference lies in the added constraint written in *italics*).

- **BRB-Validity.** If a *correct* process  $p_i$  br-delivers a message from a *correct* process  $p_j$  with sequence number  $sn$ , then  $p_j$  br-broadcasts it with sequence number  $sn$ .
- **BRB-Integrity.** For each sequence number  $sn$  and sender  $p_j$  a *correct* process  $p_i$  br-delivers at most one message with sequence number  $sn$  from sender  $p_j$ .
- **BRB-Termination-1.** If a correct process br-broadcasts a message, it br-delivers it.
- **BRB-Termination-2.** If a *correct* process br-delivers a message from a (correct or faulty) process  $p_j$ , then all correct processes br-deliver it.

It is shown in [8, 30] that  $t < n/3$  is a necessary requirement to implement BR-Broadcast. Several algorithms implementing this abstraction have been proposed. Among them, the one presented in [7] is the most famous. It works in the  $BAMP_{n,t}[t < n/3]$  model, and requires three consecutive communication steps. The one presented in [19] works in the more constrained  $BAMP_{n,t}[t < n/5]$  model, but needs only two consecutive communication steps. These algorithms show a trade-off between optimal  $t$ -resilience and time-efficiency.

## 6.2 Proof of the algorithm in $BAMP_{n,t}[t < n/3]$

The proof has the same structure, and is nearly the same, as the one for the process-crash model presented in Section 5.2.

**Notation and high-level intuition**  $\text{trf}_j^{sn}(k, v)$  now denotes a money transfer (or the associated processing event by a process) that correct processes br-deliver from  $p_j$  with sequence number  $sn$ . If  $p_j$  is a correct process, this definition is the same as the one used in the model  $CAMP_{n,t}[\emptyset]$ . If  $p_j$  is Byzantine, TRANSFER messages from  $p_j$  do not necessarily correspond to actual transfer() invocations by  $p_j$ , but the BRB-Termination-2 property guarantees that all correct processes br-deliver the *same* set of TRANSFER messages (with the same sequence numbers), and therefore agree on how  $p_j$ 's behavior should be interpreted. The reliable broadcast thus ensures a form of *weak agreement* among correct processes in spite of Byzantine failures. This weak agreement is what allows us to move almost seamlessly from a crash-failure model to a Byzantine model, with no change to the algorithm, and only a limited adaptation of its proof.

More concretely, Lemma 2 (for crash failures) becomes the next lemma whose proof is the same as for Lemma 2 in which the reference to the CBR-Termination-2 property is replaced by a reference to its BRB counterpart.

**Lemma 4.** *If a correct process  $p_i$  processes  $\text{trf}_j^{sn}(k, v)$ , then any correct process processes it.*

Similarly, Lemma 3 turns into its Byzantine counterpart, lemma 5.

**Lemma 5.** *If a correct process  $p_i$  br-broadcasts a money transfer  $\text{trf}_i^{sn}(k, v)$  (line 3), any correct processes eventually br-delivers and processes it.*

**Proof** When a correct process  $p_i$  br-broadcasts a money transfer  $\text{trf}_i^{sn}(k, v)$ , we have  $(sn = \text{del}_i[i] + 1) \wedge (\text{account}_i[i] \geq v)$ , thus when it br-delivers it the predicate of line 7 is satisfied. By Lemma 4, all the correct processes process this money transfer.  $\square_{\text{Lemma 5}}$

**Theorem 2.** *Algorithm 1 instantiated with BR-Broadcast implements a money transfer object in the system  $BAMP_{n,t}[t < n/3]$  model, and ensures that all operations by correct processes terminate.*

The model constraint  $t < n/3$  is due only to the fact that Algorithm 1 uses BR-broadcast (for which  $t < n/3$  is both necessary and sufficient). As the invocations of balance() by Byzantine processes may return arbitrary values and do not impact the correct processes, they are not required to appear in their

local histories.

**Proof** The proof that the operations issued by the correct processes terminate is the same as in Lemma 1 where the CRB-Termination properties are replaced by their BRB-Termination counterparts.

To prove MT-compliance, let us first construct mock local histories for Byzantine processes: the mock local history  $L_i$  associated with a Byzantine process  $p_j$  is the sequence of money transfers from  $p_j$  that the correct processes br-deliver from  $p_j$  and that they process. (By Lemma 4 all correct processes process the same set of money transfers from  $p_j$ ).

Let  $p_i$  be a correct process and  $S_i$  be the sequence of operations occurring at  $p_i$  defined in the same way as in the crash failure model. In this construction, the following properties are respected:

- For all,  $\text{trf}_j^{sn}(k, v) \in L_j$  then
  - if  $p_j$  is correct, it br-broadcast this money transfer and, due to Lemma 5,  $p_i$  processes it, hence  $\text{trf}_j^{sn}(k, v) \in S_i$ .
  - if  $p_j$  is Byzantine, due to the definition of  $L_j$  (sequence of money transfers that correct processes br-delivers from  $p_j$  and process), we have  $\text{trf}_j^{sn}(k, v) \in S_i$ .
- For all  $\text{op1} = \text{trf}_j^{sn}(k, v)$  and  $\text{op2} = \text{trf}_j^{sn'}(k', v')$  (two transfers in  $L_j \subseteq S_i$ ) such that  $\text{op1} \rightarrow_j \text{op2}$ , we have  $sn < sn'$ , consequently  $p_i$  processes  $\text{op1}$  before  $\text{op2}$ , and we have  $\text{op1} \rightarrow_{S_i} \text{op2}$ .
- For all both  $\text{op1}$  and  $\text{op2}$  belonging to  $L_i$ , their serialization order is the same in  $L_i$  as in  $S_i$  (same as for the crash case).

It follows that  $S_i$  is a serialization of  $A_{i,T}(\tilde{H})$  where  $\tilde{H} = (L_1, \dots, L_n)$ ,  $L_i$  being the sequence of its operations if  $p_i$  is correct, and a mock sequence of money transfers, if it is Byzantine. The same arguments that were used in the crash failure model can be used here to prove that  $S_i$  is MT-compliant. Since all correct processes observe the same mock sequence of operations  $L_j$  for any given Byzantine process  $p_j$ , it follows that the algorithm implements an MT-compliant money transfer object in  $BAMP_{n,t}[t < n/3]$ .

□*Theorem 2*

### 6.3 Extending to incomplete Byzantine networks

An algorithm is described in [31] which simulates a fully connected (point-to-point) network on top of an asynchronous Byzantine message-passing system in which, while the underlying communication network is incomplete (not all the pairs of processes are connected by a channel), it is  $(2t+1)$ -connected (i.e., any pair of processes is connected by  $(2t+1)$  disjoint paths<sup>12</sup>). Moreover, it is shown that this connectivity requirement is both necessary and sufficient.<sup>13</sup>

Hence, denoting  $BAMP_{n,t}[t < n/3, (2t+1)\text{-connected}]$  such a system model, this algorithm builds  $BAMP_{n,t}[t < n/3]$  on top  $BAMP_{n,t}[t < n/3, (2t+1)\text{-connected}]$  (both models have the same computability power). It follows that the previous money-transfer algorithm works in incomplete  $(2t+1)$ -connected asynchronous Byzantine systems where  $t < n/3$ .

## 7 Conclusion

The article has revisited the synchronization side of the money-transfer problem in failure-prone asynchronous message-passing systems. It has presented a generic algorithm that solves money transfer in asynchronous message-passing systems where processes may experience failures. This algorithm

<sup>12</sup>“Disjoint” means that, given any pair of processes  $p$  and  $q$ , any two paths connecting  $p$  and  $q$  share no process other than  $p$  and  $q$ . Actually, the  $(2t+1)$ -connectivity is required only for any pair of correct processes (which are not known in advance).

<sup>13</sup>This algorithm is a simple extension to asynchronous systems of a result first established in [11] in the context of synchronous Byzantine systems.

uses an underlying reliable broadcast communication abstraction, which differs according to the type of failures (process crashes or Byzantine behaviors) that processes can experience.

In addition to its genericity (and modularity), the proposed algorithm is surprisingly simple<sup>14</sup> and particularly efficient (in addition to money-transfer data, each message generated by the algorithm only carries one sequence number). As a side effect, this algorithm has shown that, in the crash failure model, money transfer is a weaker problem than the construction of a read/write register. As far as the Byzantine failure model is concerned, we conjecture that  $t < n/3$  is a necessary requirement for money transfer (as it is for the construction of a read/write register [18]).

Finally, it is worth noticing that this article adds one more member to the family of algorithms that strive to “unify” the crash failure model and the Byzantine failure model as studied in [6, 12, 20, 26].

## Acknowledgments

This work was partially supported by the French ANR projects 16-CE40-0023-03 DESCARTES, devoted to layered and modular structures in distributed computing, and ANR-16-CE25-0005-03 O’Browser, devoted to decentralized applications on browsers.

## References

- [1] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M., and Shavit N., Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873-890 (1993)
- [2] Ahamad M., Neiger G., Burns J.E., Hutto P.W., and Kohli P., Causal memory: definitions, implementation and programming. *Distributed Computing*, 9:37–49 (1995)
- [3] Aigner M. and Ziegler G., *Proofs from THE BOOK* (4th edition). Springer, 274 pages, ISBN 978-3-642-00856-6 (2010)
- [4] Attiya H., Efficient and robust sharing of memory in message-passing systems. *Journal of Algorithms*, 34(1):109-127 (2000)
- [5] Attiya H., Bar-Noy A., and Dolev D., Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):121-132 (1995)
- [6] Bazzi, R. and Neiger, G., Optimally simulating crash failures in a byzantine environment. *Proc. 6th Workshop on Distributed Algorithms (WDAG’91)*, Springer LNCS 579, pp. 108–128 (1991)
- [7] Bracha G., Asynchronous Byzantine agreement protocols. *Information & Computation*, 75(2):130-143 (1987)
- [8] Bracha G. and Toueg S., Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824-840 (1985)
- [9] Cachin Ch., Guerraoui R., and Rodrigues L., *Reliable and secure distributed programming*, Springer, 367 pages, ISBN 978-3-642-15259-7 (2011)
- [10] Collins D., Guerraoui R., Komatovic J., Monti M., Xygkis A., Pavlovic M., Kuznetsov P., Pignolet Y.-A., Serebinschi D.A., and Tonlikh A., Online payments by merely broadcasting messages. *Proc. 50th IEEE/IFIP Int’l Conference on Dependable Systems and Networks (DSN’20)*, 10 pages (2020)
- [11] Dolev D., The Byzantine general strike again. *Journal of Algorithms*, 3:14-30 (1982)

---

<sup>14</sup>Let us recall that, in sciences, simplicity is a first class property [3]. As stated by A. Perlis — recipient of the first Turing Award — “Simplicity does not precede complexity, but follows it”.

- [12] Dolev D. and Gafni E., Some garbage in - some garbage out: asynchronous  $t$ -Byzantine as asynchronous benign  $t$ -resilient system with fixed  $t$ -Trojan horse inputs. *Tech Report*, arXiv:1607.01210, 14 pages (2016)
- [13] Fernández Anta A., Konwar M.K., Georgiou Ch., and Nicolaou N.C., Formalizing and implementing distributed ledger objects, *SIGACT News*, 49(2):58-76 (2018)
- [14] Guerraoui R., Kuznetsov P., Monti M., Pavlovic M., Serebinschi D.A., The consensus number of a cryptocurrency. *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC'19)*, ACM Press, pp. 307-316 (2019)
- [15] Gupta S., A non-consensus based decentralized financial transaction processing model with support for efficient auditing. *Master Thesis*, Arizona State University, 83 pages (2016)
- [16] Hadzilacos V. and Toueg S., A modular approach to fault-tolerant broadcasts and related problems. *Tech Report 94-1425*, 83 pages, Cornell University (1994)
- [17] Herlihy M.P. and Wing J.M., Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492 (1990)
- [18] Imbs D., Rajsbaum S., Raynal M., and Stainer J., Read/write shared memory abstraction on top of an asynchronous Byzantine message-passing system. *Journal of Parallel and Distributed Computing*, 93-94:1-9 (2016)
- [19] Imbs D. and Raynal M., Trading  $t$ -resilience for efficiency in asynchronous Byzantine reliable broadcast. *Parallel Processing Letters*, Vol. 26(4), 8 pages (2016)
- [20] Imbs D., Raynal M., and Stainer J., Are Byzantine failures really different from crash failures? *Proc. 30th Symposium on Distributed Computing (DISC'16)*, Springer LNCS 9888, pp. 215-229 (2016)
- [21] Knuth D.E., Ancient Babylonian algorithms. *Communications of the ACM*, 15(7):671-677 (1972)
- [22] Lamport L., On interprocess communication, Part I: basic formalism; Part II: algorithms. *Distributed Computing*, 1(2):77-101 (1986)
- [23] Lamport L., Shostack R., and Pease M., The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382-401 (1982)
- [24] Mostéfaoui A. and Raynal M., Two-bit messages are sufficient to implement atomic read/write registers in crash-prone systems. *Proc. 35th ACM Symposium on Principles of Distributed Computing (PODC'16)*, ACM Press, pp. 381-390 (2016)
- [25] Nakamoto S., Bitcoin: a peer-to-peer electronic cash system. <https://bitcoin.org/bitcoin.pdf> (2008) [last accessed March 31, 2020]
- [26] Neiger, G. and Toueg, S., Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*; 11(3): 374-419 (1990)
- [27] Neugebauer O., *The exact sciences in antiquity*. Brown University press, 240 pages (1957)
- [28] Pease M., Shostak R., and Lamport L., Reaching agreement in the presence of faults. *Journal of the ACM*, 27:228-234 (1980)
- [29] Raynal M., *Concurrent programming: algorithms, principles and foundations*. Springer, 515 pages, ISBN 978-3-642-32026-2 (2013)
- [30] Raynal M., *Fault-tolerant message-passing distributed systems: an algorithmic approach*. Springer, 550 pages, ISBN: 978-3-319-94140-0 (2018)
- [31] Raynal M., From incomplete to complete networks in asynchronous Byzantine systems. *Tech report*, 10 pages (2020)



- [32] Riesen A., Satoshi Nakamoto and the financial crisis of 2008.  
<https://andrewriesen.me/2017/12/18/2017-12-18-satoshi-nakamoto-and-the-financial-crisis-of-2008/> [last accessed April 22, 2020]
- [33] Taubenfeld G., *Synchronization algorithms and concurrent programming*. Pearson Education/Prentice Hall, 423 pages, ISBN 0-131-97259-6 (2006).